Direct Model Checking of Temporal Properties[§]

R. Bharadwaj[*] J. Zucker[**]

Abstract

In this paper, we address the problem of model checking temporal properties of finite-state programs. This problem is usually solved by modelling the program as well as the *negation* of the desired temporal property as automata on infinite words (Büchi automata) and checking for emptiness of the automaton resulting from the synchronous product of the program automaton and the negated property automaton. To check properties expressed in Linear-time Temporal Logic (LTL) [MP91b] involves a construction which produces an automaton with states exponential in the size of the formula [Wol89]. This poses a problem especially when model checking under fairness assumptions — the formulae tend to be very long in these cases. In this paper, we present a method to *directly* check properties expressed in LTL for programs written in a UNITY-like notation. We do this by adapting well-known LTL proof rules to model checking. In our approach, there is no need to construct equivalent Büchi automata. Another advantage is our ability to handle fairness in a straight forward way. Our method combines model checking and theorem proving in a unified framework, and may be used for practical verifications.

1 Introduction

Model checking has emerged as a successful method to verify that a (finite-state) program satisfies its correctness requirements [CES86, Hol91, Kur89, McM93]. It is based on the idea of expressing a program's requirement as a formula in temporal logic, and viewing the program as a structure that may be interpreted as the formula's model [Wol89]. In this paper, we shall only concern ourselves with the problem of model checking formulae expressed in Linear-time Temporal Logic (LTL) [MP83, MP91b]. Other popular approaches use branching-time logics such as CTL (see for example [CES86, McM93] for details). The usual approach to model check an LTL formula is to construct a finite automaton on infinite words (a Büchi automaton) for the *negation* of the formula, take a synchronous product of the automaton representing the behaviour of the program and the Buchi automaton representing the negation of the formula, and check that the language accepted by the product automaton is empty [VW86, Wol89]. Emptiness is checked by determining whether the set of accepting states is reachable from the initial set of states, and belongs to a cycle [CVWY92, Hol91]. An advantage of this approach is that using the negation of the formula may result in a smaller state space to be explored. A popular way to implement this algorithm is to compute the program automaton and the synchronous product in the same step, obviating the need to store the whole state-space graph, an approach known as *on-the-fly (OTF)* model checking [CVWY92, Hol91].

A major disadvantage of this approach is that to model check a property expressed in LTL requires the construction of an equivalent Büchi automaton that has, in general, an exponential

[§]This work was partially funded by the Natural Sciences and Engineering Research Council of Canada and the Telecommunications Research Institute of Ontario.

[*]Code 5546, Naval Research Laboratory, Washington, DC 20375-5000, USA. E-mail: ramesh@itd.nrl.navy.mil

[**]Dept. of Computer Science, McMaster University, Hamilton, ON L8S 4K1, Canada. E-mail: zucker@mcmaster.ca

number of states in the length of the original formula [Wol89]. Also, this approach is difficult to use in practice because users, being unfamiliar with the Büchi automaton construction procedure, do not have an intuitive understanding of the generated Büchi automaton — interpreting counterexamples produced by the model checker is therefore very difficult.

Another disadvantage of this method is that fairness assumptions [Fra86, MP91b] cannot be handled very well. A plausible approach to handle fairness is to express the fairness assumption as a formula $\mathcal{F}$, and model check the formula $\mathcal{F} \Rightarrow q$, where q is the property to be checked. This approach is not very feasible because even if the original formula q expressing the desired property is short, the new formula will, in general, no longer be so. Other approaches that build fairness directly into the state exploration algorithms [Cho74, Pel93] have the disadvantage that they generate inordinately long counterexamples, making them hard to interpret.

In this paper, we explore a new approach to model check temporal properties. Our method adapts well-known proof rules used in theorem proving LTL formulae [MP91a], to model checking. Termed *direct* model checking, this approach has several advantages:

- There is no need to construct (or use) Büchi automata for model checking LTL properties.
- It unifies model checking with theorem proving methods, vastly increasing the set of tools available to users.
- Fairness (both weak fairness as well as strong fairness) is handled in a straight forward way, just as in theorem proving approaches.
- Our method is compatible with on-the-fly model checking (see section 3), and the bit-state hashing algorithm of [Hol91].

Our approach requires some intuition on the part of users, to be able to come up with auxiliary invariants (and other predicates) which may be necessary for model checking. In our opinion, this is actually an advantage because use of this approach is close to a genuine proof i.e., a chain of argument which will convince a human reviewer; the pitfalls of relying on the mere grunt of assent from an oracle (in this case, the model checking program) are well-known [RvH91].

This approach of combining theorem proving and model checking techniques, capitalises on the strengths of each to minimize the weaknesses of the other. For on the one hand, model checking algorithms inherently suffer from a major limitation, namely, state explosion[1]. We employ theorem proving as a front end to model checking, potentially reducing the state space of the problem to be model checked. We also avoid the state space explosion associated with Büchi automaton construction, as described above. On the other hand, a major barrier to the industrial use of formal verification has been that few practitioners have the mathematical sophistication or theorem proving skills required to use them. Our approach employs the automation provided by model checking to reduce the manual effort associated with pure theorem proving. Designing new verification algorithms and implementing them efficiently is very crucial for the successful industrial application of formal methods.

[1] Because they operate directly on the model, these algorithms must concretely denote all the states in the model, which may be very large when compared to the program text.

2 Preliminaries

In the following discussion, we associate a set V of *variables* with every program $\mathcal{P}$. Each variable may take on values over a (finite) domain $\mathbf{D}$. A *state* is an interpretation of V, assigning to each variable a value from domain $\mathbf{D}$.

2.1 Notation for Programs

We can represent programs as (finite) sets of transitions, as in UNITY [CM88]. Programs with constructs such as processes, channels, sequential and parallel composition, and repetition can be mechanically translated to this notation (as in [Bha95]). A program $\mathcal{P}$ is a pair $\langle \Theta, \mathcal{T} \rangle$, consisting of an *initial predicate* Θ, characterising the set of *initial states*, and a (finite) set of *transitions* $\mathcal{T}$. Each transition $\tau \in \mathcal{T}$ consists of a quantifier-free predicate, or *guard*, the symbol "$\Leftrightarrow\!\!\rightarrow$", followed by a finite set of assignments (separated by commas), called the *body*. Operationally, a transition τ is said to be *enabled* in a state s if its guard evaluates to "true" in that state. For a transition τ, we denote the set of enabled states by predicate $En(\tau)$. If a transition is enabled in state s, its body may be *executed* in that state. Assignments in the body are executed as a single multiple assignment. For example, the body "x:=y, y:=x" exchanges the values of variables x and y.

A *behaviour* of program $\mathcal{P}$ is a *linear sequence* of states $s_0, s_1, \ldots$ such that the first state s_0 of the sequence is an initial state of $\mathcal{P}$ (i.e., it satisfies the initial predicate Θ), and for each consecutive pair of states $\langle s_i, s_{i+1} \rangle$ in the sequence, there exists a transition τ of $\mathcal{P}$ such that its guard is *true* in state s_i, and state s_{i+1} results from executing τ's body in state s_i. We assume all behaviours to be **infinite** sequences, without loss of generality; for, we may extend a finite sequence ending in state s_n into an infinite one by repeating s_n infinitely often at the end of the sequence.

2.2 Notation for Properties

We assume an underlying first-order assertion language $\mathcal{L}$, with interpreted symbols for expressing the standard operations and relations over domain $\mathbf{D}$. We shall refer to formulae in language $\mathcal{L}$ as *state formulae* or *assertions*. We shall denote the proposition "true" by $\mathbf{tt}$.

We construct *temporal formulae* out of state formulae by applying the logical operators $\neg$ (negation) and $\vee$ (disjunction), and the temporal operator $\mathcal{U}$ (until). Other logical operators such as $\Rightarrow$ (implication), $\Leftrightarrow$ (equivalence), and $\wedge$ (conjunction), and temporal operators such as $\Diamond$ (eventually) and $\Box$ (henceforth) can be defined in terms of these elementary operators.

A temporal formula p is interpreted over an infinite sequence of states $\sigma : s_0, s_1, \ldots$, known as a *structure*, where each state s_i is an interpretation for the variables in p. For a structure σ, the general notion of a temporal formula p *holding* at position $j \geq 0$ in σ is written as $(\sigma, j) \models p$. In the following, we provide an inductive definition of a temporal formula p holding for sequence σ at position j:

- If p is a state formula (i.e., has no temporal operators),

$$(\sigma, j) \models p \Leftrightarrow s_j \models p$$

  we evaluate p using the interpretation given by state s_j.

- $(\sigma, j) \models \neg p \Leftrightarrow (\sigma, j) \not\models p$.

- $(\sigma, j) \models p \vee q \Leftrightarrow (\sigma, j) \models p$ or $(\sigma, j) \models q$.
- $(\sigma, j) \models p \; \mathcal{U} \; q \Leftrightarrow$ for some $k \geq j$, $(\sigma, k) \models q$ and for all i such that $j \leq i < k$, $(\sigma, i) \models p$.

In addition, we define two other temporal operators:

- $\Diamond p = \mathbf{tt} \; \mathcal{U} \; p$ (eventually)
- $\Box p = \neg \Diamond \neg p$ (henceforth).

2.3 Program Validity of LTL Formulae

The behaviour of a program $\mathcal{P}$, an infinite sequence of states, can serve as a structure over which a temporal formula p may be interpreted.

We define the validity of a temporal formula p for a program $\mathcal{P}$ by interpreting p over all behaviours of program $\mathcal{P}$. If p is valid for all behaviours of program $\mathcal{P}$, we say program $\mathcal{P}$ *satisfies* the temporal property p. We may check this by using proof rules for establishing temporal properties of programs. In this paper, we outline how these proof rules may be adapted for model checking, and propose an algorithm to accomplish this task.

We shall restrict ourselves to the following groups of properties[2]. These proof rules suffice to establish most temporal properties, and serve as working tools during practical verifications [MP90].

One group of rules establishes the validity of the *invariance* formulae $\Box q$ and $\Box(p \Rightarrow \Box q)$; these formulae express the invariance of a state formula q, either throughout a behaviour, or starting from the state in a behaviour in which formula p holds.

Another group of rules establishes the validity of the *eventuality* formulae $\Diamond q$ and $\Box(p \Rightarrow \Diamond q)$; these formulae express the guarantee that q will eventually be true in a behaviour, either once, or following each state in which formula p holds.

These proof rules reduce the problem of establishing an LTL property for a program to that of establishing the validity of a (finite) set of state formulae. Some of these are directly expressed as (first-order) predicates. Others, known as *annotated transitions*, are of the form $\{pre\}\tau\{post\}$, where *pre* and *post* are predicates and τ is a transition. The interpretation of $\{pre\}\tau\{post\}$ is that whenever the guard of τ is true at a state in which *pre* holds, then *post* holds at the state resulting from the execution of the body of τ.

We may translate each annotated transition into a first-order predicate. We do this by using the notion of *weakest precondition* [Dij76]. Let B denote the body $x_1 := e_1, \cdots, x_n := e_n$. The weakest precondition $wp(B, post)$, where *post* is a predicate, is defined as $post[e_1/x_1, \cdots, e_n/x_n]$, where $post[e_1/x_1, \cdots, e_n/x_n]$ denotes the predicate resulting from the simultaneous substitution of e_i for x_i in *post* $(i = 1, \cdots, n)$. An annotated transition $\{pre\}\tau\{post\}$, where $\tau = g \Leftrightarrow B$, is equivalent to the predicate $(pre \wedge g) \Rightarrow wp(B, post)$.

2.4 Notation for Annotated Programs

An *annotated program* $\mathcal{P}^A$ is a pair $\langle \Theta^A, \mathcal{T}^A \rangle$. It consists of an *annotated initial predicate* Θ^A and a (finite) set of annotated transitions $\mathcal{T}^A$. $\Theta^A = \langle \Theta, Init \rangle$, where Θ and *Init* are predicates, is written as $\Theta\{Init\}$. Each annotated transition $\tau^A \in \mathcal{T}^A$ is a triple $\langle pre, \tau, post \rangle$ written as $\{pre\}\tau\{post\}$.

[2]The proof system may be extended to a richer system, which establishes validity of any LTL formula; for details see [MP91a, MP91b].

Here, *pre* and *post* are predicates and τ is a transition of the form $g \Leftrightarrow B$ with guard g and body B. We define the enabledness of an annotated transition τ^A in terms of the enabledness of its component transition τ.

Operationally, we may define a behaviour of an annotated program just as we did for a program: a behaviour $\sigma = s_0, s_1, \ldots$ starts from a state s_0 satisfying predicate Θ of Θ^A, and for each pair of consecutive states s_i and s_{i+1} in σ, execution of the transition part of an enabled annotated transition in state s_i results in state s_{i+1}.

Additionally, each behaviour must satisfy the following conditions:

- Predicate *Init* must hold at state s_0. If not, the behaviour is said to have an *assertion violation* in state s_0. In this event, the singleton sequence s_0 is called a *counterexample*.
- If state s_{i+1} results from state s_i by executing the transition part of an enabled annotated transition τ^A, and the predicate part *pre* of τ^A holds in state s_i, then predicate part *post* of τ^A must hold in state s_{i+1}. If not, such a behaviour is said to have an *assertion violation* in state s_i. In such a case, and additionally if there are no assertion violations in states $s_0, \ldots, s_{i-1}$, the sequence of states $s_0, \ldots s_i$ is called a *counterexample*.

3 The Model Checking Algorithm

The model checking problem we consider is as follows: given an annotated program $\mathcal{P}^A$ check for assertion violations, if any, and provide a counterexample for the first encountered violation. We call this process *assertion checking*. The algorithm, adapted from [HS82], performs a depth-first search (DFS) of the behaviours of program $\mathcal{P}^A$. In the following, the notation $Pred(s)$ where *Pred* is a predicate and s is a state is used to denote the evaluation of predicate *Pred* in state s (yielding a boolean result). We denote the cardinality of set $\mathcal{T}^A$ by $\mid \mathcal{T}^A \mid$. We use the notation τ_i^A to denote the $i^{\texttt{th}}$ annotated transition of $\mathcal{T}^A$. The components of τ_i^A are respectively denoted by $\tau_i^A.pre$, τ_i, and $\tau_i^A.post$. Predicate $enabled(\tau_i^A, s)$ denotes the enabledness condition of transition τ_i^A in state s. Function $succ(\tau, s)$ returns the state resulting from the execution of transition τ in state s.

$H := \emptyset$; $S :=$ `empty`
for each s_0 such that $\Theta(s_0)$ and $s_0 \notin H$ **do**
  $cs := s_0$; $assert(Init(s_0))$; $H := H \cup \{s_0\}$; $push(S, \langle s_0, 1\rangle)$;
  while $S \neq$ `empty` **do**
    $\langle cs, i\rangle := top(S)$; $pop(S)$
    while $i \leq \mid \mathcal{T}^A \mid$ **do**
      if $enabled(\tau_i^A, cs)$ **then**
        $ns := succ(\tau_i, cs)$
        if $\tau_i^A.pre(cs)$ **then** $assert(\tau_i^A.post(ns))$ **end**
        if $ns \notin H$ **then**
          $H := H \cup \{ns\}$; $push(S, \langle cs, i+1\rangle)$;
          $cs := ns$; $i := 1$
        else $i := i + 1$ **end**
      else $i := i + 1$ **end**
    end
  end
end

The main data structures used are the following: Stack S contains the sequence of states of a behaviour, starting from the initial state up to (and excluding) the *current state* (cs). Elements of stack S are pairs $\langle s, n \rangle$ where $s \in \mathbf{D}$ is a state and $n \in \mathbf{Nat}$ is a natural number denoting the first unexplored transition of state s. We denote the empty stack by `empty`; stack operations *push*, *pop*, and *top* have the usual meaning. Set H holds all states that have been visited during the DFS. Set H is usually implemented as a hash-table. Set operations $\cup$ and $\in$ have the usual meaning. The algorithm uses an auxiliary procedure $assert(p)$, which prints out the state component of elements of stack S, and state cs, if predicate p is false.

4 Adapting LTL Proof Rules to Model Checking

In this section, we outline how we may interpret LTL proof rules in the context of model checking. We have adapted these proof rules from [MP83, MP90, MP91a, MP91b]. As we mentioned in section 2.3, we shall restrict ourselves to proof rules for invariance and eventuality formulae, as our intent is to convey to the reader our central idea. It should be easy to extend our work to the general case of model checking any LTL formula (on the lines of [MP91a, MP91b]).

Using a proof rule, one may infer an LTL property for a program from the validity of a set of formulae known as the *premises*. In this section, we present two versions of a number of proof rules. One version is meant for carrying out proofs using the classical approach, in which these validities are established individually by providing a proof in a formal axiomatic theory. The second version is meant for the direct model checking approach: in this approach, the validity of the premises is established either by translating them into model checking sub-problems (using auxiliary proof rules), or by collectively establishing validity of all premises by assertion checking an annotated program using the algorithm outlined in section 3.

In the following, we denote by $\{pre\}T\{post\}$, where *pre* and *post* are predicates, and $T \subseteq \mathcal{T}$ is a set of transitions, the condition of requiring $\{pre\}\tau\{post\}$ to hold for every $\tau \in T$.

4.1 Rules for Invariance

4.1.1 Example 1: Establishing $\Box q$

(a) Proof Rule for $\Box q$: For a program $\mathcal{P}$, a proof rule to establish the validity of temporal formula $\Box q$, where q is a state formula, is as below.

INV	I1.	$\Theta \Rightarrow \phi$
	I2.	$\phi \Rightarrow q$
	I3.	$\{\phi\}\mathcal{T}\{\phi\}$
		$\Box q$

This rule uses an auxiliary predicate ϕ to establish the invariance of q. We can prove its soundness by showing that ϕ holds for all states s_j of a behaviour, by induction on j, using premises I1 and I3. From this and premise I2 the conclusion follows.

(b) Model Checking Rule for $\Box q$: For a program $\mathcal{P}$, the validity of temporal formula $\Box q$, where q is a state formula, is established by the following *model checking rule INV**:

INV^*	Θ^A:	$\Theta\{q\}$
	$\mathcal{T}^A$:	$\{q\}\mathcal{T}\{q\}$
	$\mathcal{P} \models$	$\Box q$

This rule asserts that in order to establish, for program $\mathcal{P} = \langle \Theta, \mathcal{T} \rangle$, the invariance of a state property q (i.e., $\Box q$), it is sufficient to assertion check an annotated program $\mathcal{P}^A$ with annotated initial predicate $\Theta\{q\}$, and annotated transitions $\{q\}\mathcal{T}\{q\}$.

We prove its soundness by establishing the premises of proof rule **INV**, by setting $\phi = q$. From this follows the conclusion of **INV**, and hence the conclusion of INV^*.

Note that in this *model checking* approach, we *collectively* establish *all* the premises of rule **INV** essentially by evaluating predicate q in every reachable state of program $\mathcal{P}$. Because we evaluate predicate q only in the reachable states of program $\mathcal{P}$, an auxiliary (stronger) predicate ϕ is not required in this approach. A detailed explanation is given in [Bha95].

4.1.2 Example 2: Establishing $\Box(p \Rightarrow \Box q)$

(a) Proof Rule for $\Box(p \Rightarrow \Box q)$: For a program $\mathcal{P}$, a proof rule to establish the validity of temporal formula $\Box(p \Rightarrow \Box q)$, where p and q are state formulae, is given below (**CINV** stands for "conditional invariance").

CINV	C1.	$p \Rightarrow \phi$
	C2.	$\phi \Rightarrow q$
	C3.	$\{\phi\}\mathcal{T}\{\phi\}$
		$\Box(p \Rightarrow \Box q)$

As in rule **INV**, rule **CINV** requires an auxiliary predicate ϕ to establish the invariance of q, starting from a state s_i in which predicate p holds. Its soundness can be easily proved as before.

(b) Model Checking Rule for $\Box(p \Rightarrow \Box q)$: To model check a formula of the form $\Box(p \Rightarrow \Box q)$, where p and q are state formulae, we first formulate the following auxiliary proof rule:

CINVa	C1a.	$\Box(p \Rightarrow q)$
	C2a.	$\Box(q \Rightarrow \Box q)$
		$\Box(p \Rightarrow \Box q)$

Soundness of this rule follows from the rule of *entailment transitivity*: $\Box(p \Rightarrow q), \Box(q \Rightarrow r) \vdash \Box(p \Rightarrow r)$ (see [MP91b] page 230).

In practice, premise C1a is established either by theorem proving (e.g., by establishing the validity of $p \Rightarrow q$), or by showing that it is an invariant property of program $\mathcal{P}$ using the model checking rule INV^*.

Premise C2a, i.e., a property of the form $\Box(q \Rightarrow \Box q)$ is known as a *stability* property. This formula asserts that if q holds at any state in a behaviour, then it will continue to hold for the rest of the behaviour. We may establish this either by using proof rule **CINV** or by the following model checking rule $CINV^*$:

*CINV**	Θ^A:	$\Theta\{\mathbf{tt}\}$
	$\mathcal{T}^A$:	$\{q\}\mathcal{T}\{q\}$
	$\mathcal{P} \models$	$\Box(q \Rightarrow \Box q)$

This rule asserts that in order to establish that a state property q is stable for program $\mathcal{P} = \langle \Theta, \mathcal{T} \rangle$, it is sufficient to assertion check an annotated program $\mathcal{P}^A$ with annotated initial predicate $\Theta\{\mathbf{tt}\}$ (where **tt** stands for the predicate "true"), and annotated transitions $\{q\}\mathcal{T}\{q\}$.

To prove soundness: Assume that predicate q holds at state s_i of a behaviour of program $\mathcal{P}$. We can show that predicate q will continue to hold for all states s_j $(j \geq i)$ of the behaviour by induction on j.

4.2 Rules for Eventuality

4.2.1 Example 3: Basic Response under Weak Fairness

(a) Proof Rule for $\Box(p \Rightarrow \Diamond q)$: This formula asserts that every state s_i where formula p holds is followed by a state s_j $(j \geq i)$ where formula q holds. Response properties are usually established under a *fairness assumption* [Fra86, MP91b]. A common fairness assumption, known as *weak fairness*, formalises the concept of "finite progress" [Dij68] — if a program $\mathcal{P}$ has an enabled operation, then it will be eventually executed. The effect of a fairness assumption is to disallow behaviours that would otherwise be legal behaviours of program $\mathcal{P}$. The following rule relies on weak fairness to ensure that a helpful transition τ_k, leading to q, will be taken.

RESP-W	R-W1.	$\Box(p \Rightarrow (q \vee \phi))$
	R-W2.	$\{\phi\}\mathcal{T}\{q \vee \phi\}$
	R-W3.	$\{\phi\}\tau_k\{q\}$
	R-W4.	$\Box(\phi \Rightarrow En(\tau_k))$
		$\Box(p \Rightarrow \Diamond q)$

We prove soundness by contradiction. Assume that p holds at state s_i, but q does not hold at states s_j $(j \geq i)$. Then, ϕ must hold at all states s_j $(j \geq i)$ (by premises R-W1, R-W2). Therefore, transition τ_k will never be taken (by premise R-W3). This means that transition τ_k is continuously enabled in these states but never taken (by premise R-W4), which violates our fairness assumption.

(b) Model Checking Rule for $\Box(p \Rightarrow \Diamond q)$: In order to adapt rule **RESP-W** to model checking, we first formulate the following auxiliary proof rule **RESP-Wa**:

RESP-Wa	R-W1a.	$\Box(p \Rightarrow \phi)$
	R-W2a.	$\Box(\phi \Rightarrow \Diamond q)$
		$\Box(p \Rightarrow \Diamond q)$

Soundness of this rule follows again from entailment transitivity.

To model check a basic response formula, we first apply proof rule **RESP-Wa**. Premise R-W1a can be established by using model checking rule *INV** or by using a combination of model checking

and theorem proving methods. Premise R-W2a is established by the following model checking rule $RESP \Leftrightarrow W^*$:

$RESP \Leftrightarrow W^*$	Θ^A:	$\Theta\{\mathbf{tt}\}$
	$\mathcal{T}_1^A$:	$\{\phi\}\mathcal{T}\{\phi \wedge En(\tau_k)\}$
	$\mathcal{T}_2^A$:	$\{\phi\}\tau_k\{q\}$
	$\mathcal{P} \models$	$\Box(\phi \Rightarrow \Diamond q)$

To prove soundness, assume that ϕ holds at state s_i. Then, ϕ holds and τ_k is enabled at all states s_j $(j \geq i)$ (by premise $\mathcal{T}_1^A$). Therefore, by weak fairness, for some $j \geq i$, τ_k is taken at s_j and hence q holds at state s_{j+1} (by premise $\mathcal{T}_2^A$). This gives the conclusion.

Note that program $RESP \Leftrightarrow W^*$ contains annotated transitions $\{\phi\}\tau_k\{\phi \wedge En(\tau_k)\}$ **and** $\{\phi\}\tau_k\{q\}$ (since $\tau_k \in \mathcal{T}$). Our model checking algorithm has been designed in such a way that assertions of both transitions will be checked. An important point is that *behaviours that violate the weak fairness assumption will be generated during model checking.* However, these behaviours will not trigger assertion violations, as we only check for violations under the assumption of weak fairness.

4.2.2 Example 4: Basic Response under Strong Fairness

(a) Proof Rule for $\Box(p \Rightarrow \Diamond q)$: Let us now examine how the basic response property may be established under the assumption of *strong fairness*[3]. The following rule relies on strong fairness to ensure that a helpful transition τ_k, leading to q, will be taken.

RESP-S	R-S1.	$\Box(p \Rightarrow (q \vee \phi))$
	R-S2.	$\{\phi\}\mathcal{T}\{q \vee \phi\}$
	R-S3.	$\{\phi\}\tau_k\{q\}$
	R-S4.	$\Box(\phi \Rightarrow \Diamond(q \vee En(\tau_k)))$
		$\Box(p \Rightarrow \Diamond q)$

The difference between this rule and the previous rule is the fourth premise. Premise R-W4 requires that ϕ holding at a state s_i implies that the helpful transition τ_k is enabled in state s_i. Premise R-S4, however, only requires formula q to hold or transition τ_k to be enabled in a state s_j, $(j \geq i)$.

We prove soundness by contradiction. Assume that p holds at state s_i, and q does not hold at states s_j, $(j \geq i)$. Then ϕ holds continuously, with transition τ_k never being taken (by premises R-S1 – R-S3). Hence, transition τ_k is enabled infinitely many times (by premise R-S4), which violates our assumption of strong fairness.

(b) Model Checking Rule for $\Box(p \Rightarrow \Diamond q)$: The above rule **RESP-S** may be adapted to model checking on the lines of rule **RESP-W**. The fourth premise, an eventuality formula, is established either by theorem proving or by using model checking rule $RESP \Leftrightarrow W^*$.

[3] Intuitively, the requirement of strong fairness disallows computations in which a transition τ is enabled infinitely many times but taken only finitely many times.

4.2.3 Example 5: Basic Response using Well-Founded Induction

Let us now examine how we may establish eventuality properties without relying on single helpful transitions. Known as *extended response properties*, they are established based on the principle of *well-founded induction*, by an argument similar to proofs of termination for sequential programs.

Let $\prec$ be a partial order on a set $\mathcal{A}$. The structure $(\mathcal{A}, \prec)$ is said to be *well-founded* if there is no infinite sequence $a_0, a_1, a_2, \ldots$ $(a_i \in \mathcal{A})$ such that for $i = 0, 1, 2, \ldots$, $a_{i+1} \prec a_i$. The following rule uses *well-founded induction* to establish eventuality properties. Here δ is a term and z is a *rigid variable*[4]. Typically $(\mathcal{A}, \prec)$ is the naturals with the standard ordering.

RESP-WF

R-WF1. $\Box(p \Rightarrow (q \vee \phi))$

R-WF2. $\Box((\phi \wedge (\delta = z)) \Rightarrow \Diamond(q \vee (\phi \wedge (\delta \prec z))))$

$\Box(p \Rightarrow \Diamond q)$

Soundness is proved as follows: If predicate p holds at state s_i, then either q or ϕ hold at state s_i (by premise R-WF1). If ϕ holds and $\delta = z$, then eventually we will reach a state in which either q holds, or ϕ is maintained but with δ smaller than z (by premise R-WF2). Since $(\mathcal{A}, \prec)$ is well-founded, the value of δ cannot keep decreasing indefinitely. Therefore, predicate q must hold eventually.

This principle can also be used in model checking, as follows. Premise R-WF1, an invariance property, can be established as outlined in section 4.1. Premise R-WF2, an eventuality formula, is usually established by one of the two previous rules.

5 Conclusion and Ongoing Work

We have presented a direct model checking approach for checking temporal properties of finite-state programs. The traditional approach requires the construction of equivalent automata on infinite words (Büchi automata), and checking for their emptiness. Such a construction results in an automaton with an exponential number of states in the length of the original formula, which is a problem especially with model checking under fairness assumptions. Another drawback is the difficulty in interpreting counterexamples produced by the model checker.

A well known problem of model checking is that it does not supply an argument which will convince a reviewer — the fact that a model checking algorithm has detected no counterexamples to a temporal claim is not sufficient evidence to convince a reviewer of its validity. When providing a proof of validity, theorem proving systems which embed model checking algorithms, e.g., [MAB+94, RSS95] state only that a model checking algorithm was invoked. The approach outlined in this paper brings model checking closer to theorem proving, in that one may extract a chain of arguments as a result of the model checking effort, allowing a tight integration of model checking algorithms with deductive rules of inference within a theorem prover. Our method also allows portions of a model checking problem to be verified by theorem proving methods. This increases the array of tools at the disposal of users.

At the time of writing, we have a prototype implementation of the model checker available for experimentation. We have been using it for practical verifications, notably for the analysis of

[4]Rigid variables are those which are introduced in the course of a proof as logical variables, and not as translations of program variables; a rigid variable must have the same value in all states of the behaviour of a program [MP91b].

liveness violations in the DQDB protocol [HCM92]. Other applications include consistency checking of SCR-style *requirements specification documents* [AFB+88, BHJ96, PM91].

Work is also under way to integrate this model checking tool into *Snap*, a theorem proving system [Bha95, BS94]. Carrying out model checking in the context of such a system will enable users to manage complex verifications by performing reductions and transformations as described above.

References

[AFB+88] T. Alspaugh, S. Faulk, K. Britton, R. Parker, D. L. Parnas. *Software Requirements for the A-7E Aircraft.* Naval Research Laboratory, March 1988.

[Bha95] R. Bharadwaj. *Tools to Support a Formal Verification Method for Systems with Concurrency and Nondeterminism.* Ph.D. Thesis, Communications Research Laboratory, McMaster University, Hamilton, Ontario, Canada 1995.

[BHJ96] R. Bharadwaj, C. Heitmeyer, R. Jeffords. *"Reasoning about SCR Specifications."* In *Proc. 5th Int'l Software Cost Reduction Workshop.* Bell-Northern Research Ltd., Ottawa, Ontario Canada, Feb 1996 (to appear).

[BS94] R. Bharadwaj, F. A. Stomp. *Towards a Checker/Annotator for Proofs of Distributed Programs.* Technical Memorandum, AT&T Bell Laboratories, Murray Hill, NJ 1994.

[CES86] E. M. Clarke, E. A. Emerson, A. P. Sistla. *"Automatic verification of finite-state concurrent systems using temporal logic specifications."* ACM TOPLAS, 8(2): 244–263, 1986.

[CM88] K. M. Chandy, J. Misra. *Parallel Program Design – A Foundation.* Addison-Wesley, 1988.

[Cho74] Y. Choueka. *"Theories of automata on ω-tapes: a simplified approach."* *Journal of Computer and System Science 8*, 117–141, 1974.

[CVWY92] C. Courcoubetis, M. Vardi, P. Wolper, M. Yannakakis. *"Memory efficient algorithms for the verification of temporal properties."* In *Formal Methods in System Design*, 1: 275–288, Kluwer Academic Publishers, 1992.

[Dij68] E. W. Dijkstra. *"Cooperating Sequential Processes."* In *Programming Languages*, F. Genuys (Ed.), Academic Press, NY 1968.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[Fra86] N. Francez. *Fairness.* Springer-Verlag, 1986.

[HCM92] E. L. Hahne, A. K. Choudhury, N. F. Maxemchuk. *"DQDB networks with and without bandwidth balancing."* IEEE Trans. on Communications, 40(7): 1192–1204, 1992.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[HS82] E. Horowitz, S. Sahni. *Fundamentals of Data Structures.* Computer Science Press 1984.

[Kur89] R. P. Kurshan. *"Analysis of discrete event coordination."* LNCS 430, pp. 414–453, Springer-Verlag 1989.

[MAB+94] Z. Manna et al. *STeP: The Stanford Temporal Prover.* Technical Report STAN-CS-TR-94-1518, Computer Science Department, Stanford University, July 1994.

[McM93] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers 1993.

[MP83] Z. Manna, A. Pnueli. *"Verification of concurrent programs: a temporal proof system."* Foundations of Computer Science IV, Distributed Systems: Part 2, Mathematical Centre Tracts 159, pp. 163–255, Center for Mathematics and Computer Science 1983.

[MP90] Z. Manna, A. Pnueli. *"Tools and rules for the practicing verifier."* In *Carnegie Mellon Computer Science: A 25-year Commemorative*, ACM Press 1990.

[MP91a] Z. Manna, A. Pnueli. *"Completing the temporal picture."* Theoretical Computer Science, 83(1): 97–130, 1991.

[MP91b] Z. Manna, A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag 1991.

[PM91] D. L. Parnas, J. Madey. *Functional Documentation for Computer Systems Engineering.* T.R. 237, Communications Research Laboratory, McMaster University, Hamilton, Ontario, Canada September 1991.

[Pel93] D. Peled. *"All from one, one for all: on model checking using representatives."* In *Proc. 5^{th} Workshop on Computer Aided Verification*, Springer-Verlag 1993.

[RSS95] S. Rajan, N. Shankar, M. K. Srivas. *"An integration of model-checking with automated proof checking."* In *Proc. 7^{th} International Workshop on Computer-Aided Verification*, Springer-Verlag 1995.

[RvH91] J. Rushby, F. von Henke. *Formal verification of algorithms for critical systems.* Computer Science Laboratory, SRI International 1991.

[VW86] M. Y. Vardi, P. Wolper. *"An automata-theoretic approach to automatic program verification."* In *Symp. Logic in Computer Science*, Cambridge MA, pp. 322–331, 1986.

[Wol89] P. Wolper. *"On the relation of programs and computations to models of temporal logic."* In *Proceedings of Temporal Logic in Specification*, LNCS 398 B. Banieqbal, H. Barringer and A. Pnueli (Eds.), pp. 75–123, Springer-Verlag 1989.